HeLx

Release 1.0

Mar 24, 2021

Contents

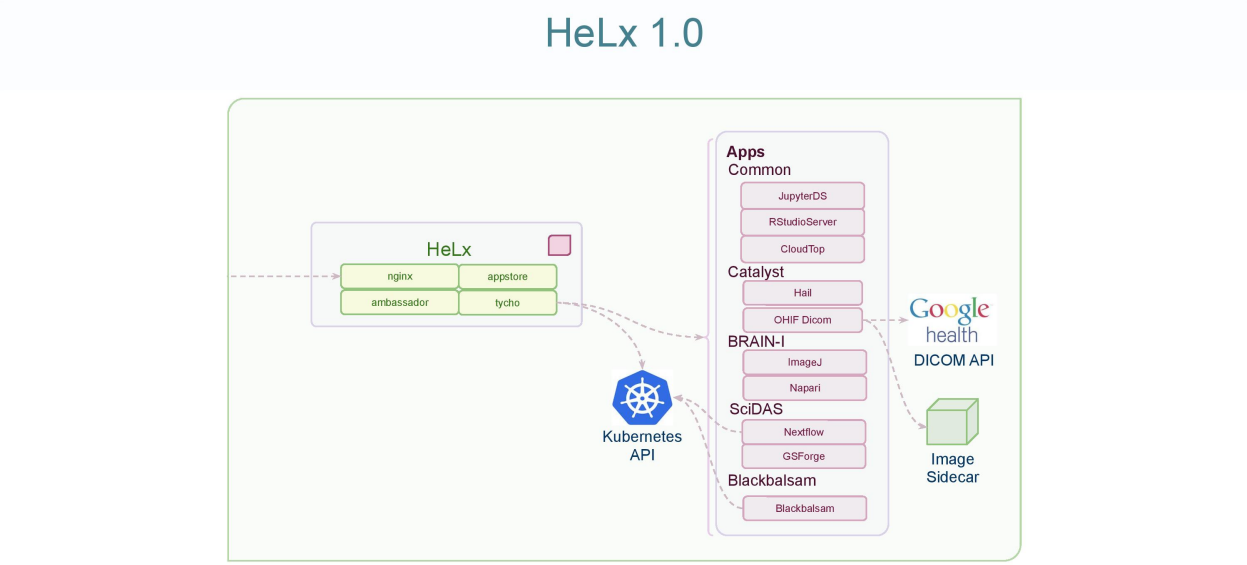
Release 1.0

Date Mar 24, 2021

CHAPTER 1

Architecture

HeLx puts the most advanced analytical scientific models at investigator's finger tips using equally advanced cloud native, container orchestrated, distributed computing systems. HeLx can be applied in many domains. Its ability to empower researchers to leverage advanced analytical tools without installation or other infrastructure concerns has broad reaching benefits.



CHAPTER 2

BRAIN-I

BRAIN-I investigators create large images of brain tissue which are then visualized and analyzed using a variety of tools which, architecturally, are not web applications but traditional desktop environments. These include image viewers like ImageJ and Napari. AppStore presents these kinds of workspaces using CloudTop, a Linux desktop with screen sharing software and adapters for presenting that interface via a web browser. CloudTop allows us to create HeLx apps for ImageJ, Napari, and other visualization tools. These tools would be time consuming, complex, and error prone, for researchers to install and would still require them to acquire the data. With CloudTop, the system can be run colocated with the data with no installation required.

CHAPTER 3

SciDAS

The Scientific Data Analysis at Scale project brings large scale computational workflow for research to cloud and on premise computing. Using the AppStore, users are able to launch Nextflow API, a web based user interface to the Nextflow workflow engine. Through that interface and associated tools, they are able to stage data into the system through a variety of protocls, execute Nextflow workflows such as the GPU accelerated KINK workflow. AppStore and associated infrastructure has run KINK on the Google Kubernetes Engine and is being installed on the Nautilus Optiputer.

CHAPTER 4

BioData Catalyst

NHLBI BioData Catalyst is a cloud-based platform providing tools, applications, and workflows in secure workspaces. The RENCI team participating in the program uses HeLx as a development environment for new applications. It is the first host for the team's DICOM medical imaging viewer. The system is designed to operate over 11TB of images in the cloud. We have created versions using the OrthaNC DICOM server at the persistence layer as well as the Google Health Dicom API service. HeLx also serves as the proving ground for concepts and demonstrations contributed to the BDC Apps and Tools Working Group. For more information, see the BioData Catalyst website.

CHAPTER 5

Blackbalsam

Blackbalsam is an open source data science environment with an initial focus on COVID-19 and North Carolina. It also serves as an experimental space for ideas and prototypes, some of which will graduate into the core of HeLx. For more information, see the blackbalsam documentation.

CHAPTER 6

User Experience

Users browse tools of interest and launch those tools to explore and analyze available data. From the user's perspective, HeLx will feel like an operating system since it runs useful tools. But there's no installation, the data's already present in the cloud with the computation, and analyses can be reproducibly shared with others on the platform.

Contact HeLx Help with questions.

6.1 HeLx Framework Overview

6.1.1 Tycho

Tycho is an API, compiler, and executor for cloud native distributed systems.

- A subset of docker-compose is the system specification syntax.
- Kubernetes is the first supported orchestrator.
- The Helm chart for deploying Tycho can be found here.

6.1.1.1 Goals

- **Application Simplity**: The Kubernetes API is reliable, extensive, and well documented. It is also large, complex, supports a range of possibilities greater than many applications need, and often requires the creation and control of many objects to execute comparatively simple scenarios. Tycho bridges the simplicity of Compose to the richness of the Kubernetes' architecture.
- **Microservice**: We wanted an end to end Python 12-factory style OpenAPI microservice that fits seamlessly into a Python ecosystem (which is why we did not use the excellent Kompose tool as a starting point).
- **Lifecycle Management**: Tycho treats distributed systems as programs whose entire lifecycle can be programmatically managed via an API.
- **Pluggable Orchestrators**: The Tycho compiler abstracts clients from the orchestrator. It creates an abstract syntax tree to model input systems and generates orchestrator specific artifacts.

- **Policy**: Tycho now generates network policy configurations governing the ingress and egress of traffic to systems. We anticipate generalizing the policy layer to allow security and other concerns to be woven into a deployment dynamically.

6.1.1.2 Partnering Technologies

This work relies on these foundations:

- PIVOT: A cloud agnostic scheduler with an API for executing distributed systems.
- Kubernetes: Widely deployed, highly programmable, horizontally scalable container orchestration platform.
- Kompose: Automates conversion of Docker Compose to Kubernetes. Written in Go, does not provide an API. Supports Docker Compose to Kubernetes only.
- Docker: Pervasive Linux containerization tool chain enabling programmable infrastructure and portability.
- Docker-compose: Syntax and tool chain for executing distributed systems of containers.
- Docker Swarm: Docker only container orchestration platform with minimal adoption.

6.1.1.3 Quick Start

samples/jupyter-ds/docker-compose.yaml:

```
---
# Docker compose formatted system.
version: "3"
services:
  jupyter-datascience:
    image: jupyter/datascience-notebook
    entrypoint: start.sh jupyter lab --LabApp.token=
    ports:
      - 8888:8888
```

In one shell, run the API:

```
$ export PATH=~/dev/tycho/bin:$PATH
$ tycho api --debug
```

In another, launch three notebook instances.

```
$ export PATH=~/dev/tycho/bin:$PATH
$ tycho up -f sample/jupyter-ds/docker-compose.yaml
SYSTEM                          GUID                                    PORT
jupyter-ds                      909f2e60b83340cd905ae3865d461156    32693
$ tycho up -f sample/jupyter-ds/docker-compose.yaml
SYSTEM                          GUID                                    PORT
jupyter-ds                      6fc07ab865d14c4c8fd2d6e0380b270e    31333
$ tycho up -f sample/jupyter-ds/docker-compose.yaml
SYSTEM                          GUID                                    PORT
jupyter-ds                      38f01c140f0141d9b4dc1baa33960362    32270
```

Then make a request to each instance to show it's running. It may take a moment for the instances to be ready, especially if you're pulling a container for the first time.

```
$ for p in $(tycho status | grep -v PORT | awk '{ print $4 }'); do
   url=http://$(minikube ip):$p; echo $url; wget -q -O- $url | grep /title;
done
http://192.168.99.111:32270
  <title>JupyterLab</title>
http://192.168.99.111:31333
  <title>JupyterLab</title>
http://192.168.99.111:32693
  <title>JupyterLab</title>
```

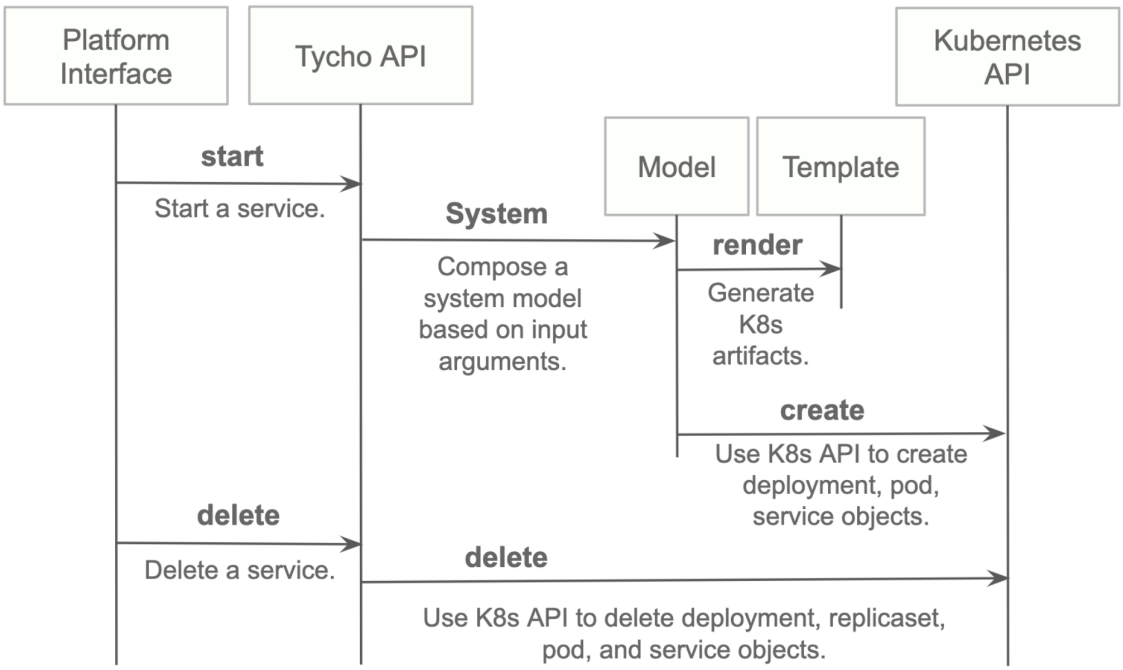
Delete all running deployments.

```
$ tycho down $(tycho status --terse)
38f01c140f0141d9b4dc1baa33960362
6fc07ab865d14c4c8fd2d6e0380b270e
909f2e60b83340cd905ae3865d461156
```

And show that they're gone.

```
$ tycho status
None running
```

6.1.1.4 Architecture



6.1.1.5 Install

- Install python 3.7.x or greater.
- Create a virtual environment.
- Install the requirements.

- Start the server.

```
python3 -m venv environmentName
source environmentName/bin/activate
pip install -r requirements.txt
export PATH=<tycho-repo-dir>/bin:$PATH
tycho api
```

6.1.1.6 Usage - A. Development Environment Next to Minikube

This mode uses a local minikube instance with Tycho running outside of Minikube. This is the easiest way to add and test new features quickly.

Run minikube:

```
minikbue start
```

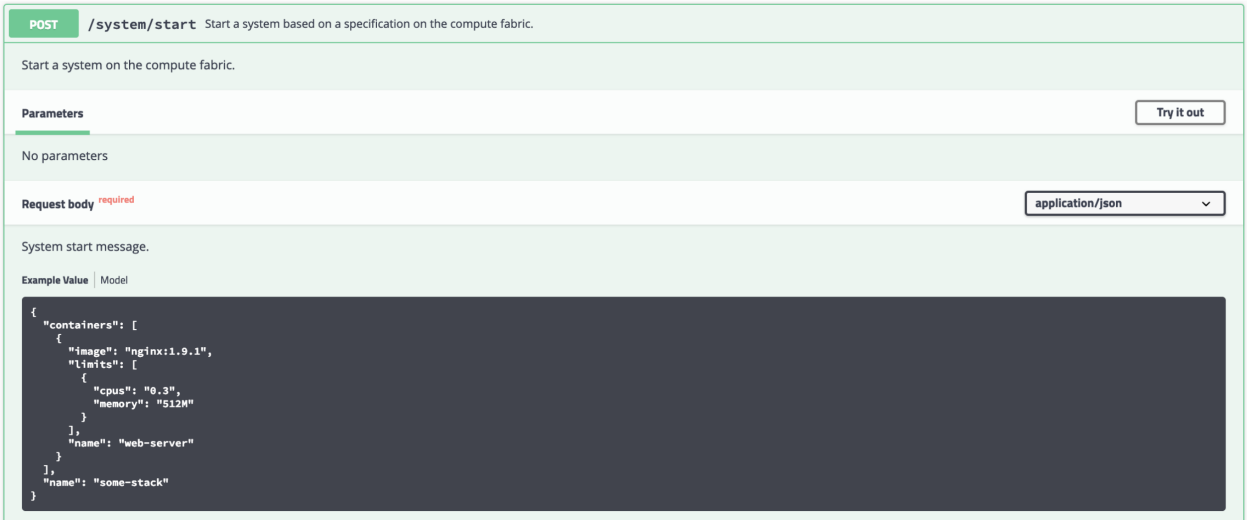
Run the minikube dashboard:

```
minikube dashboard
```

Run the Tycho API:

```
cd tycho
PYTHONPATH=$PWD/.. python api.py
```

Launch the Swagger interface `http://localhost:5000/apidocs/`



Use the Tycho CLI client as shown above or invoke the API.

Usage - B. Development Environment Within Minikube

When we deploy Tycho into Minikube it is now able to get its Kubernetes API configuration from within the cluster.

In the repo's kubernetes directory, we define deployment, pod, service, clusterrole, and clusterrolebinding models for Tycho. The following interaction shows deploying Tycho into Minikube and interacting with the API.

We first deploy all Kubernetes Tycho-api artifacts into Minkube:

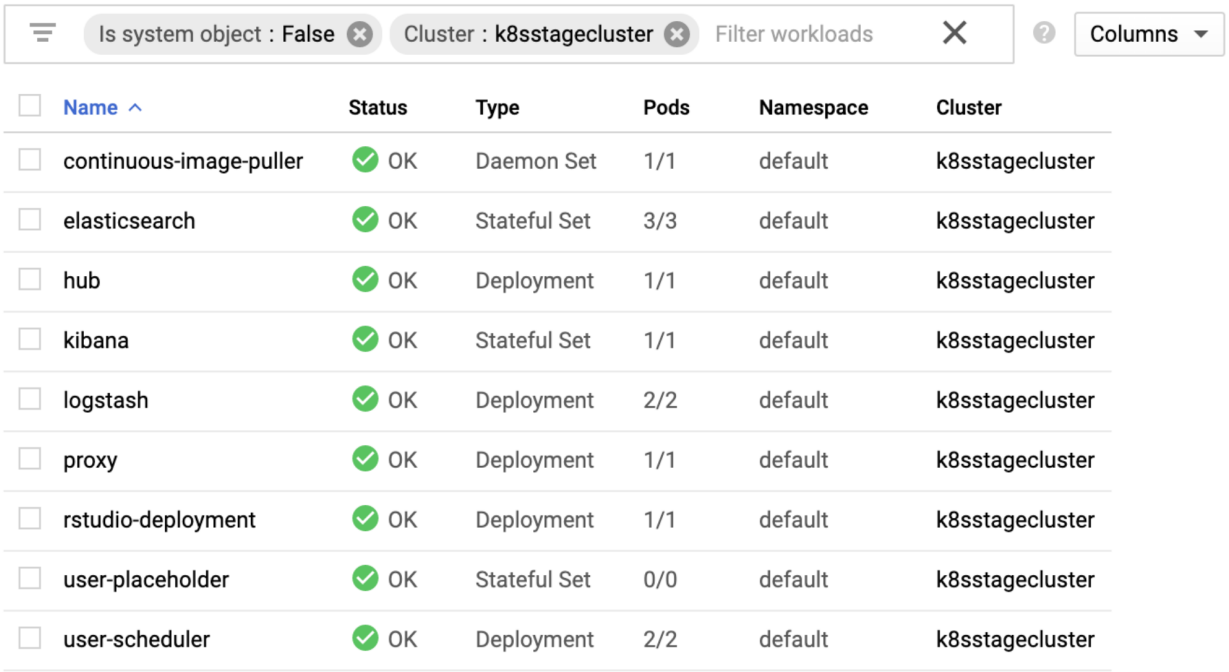
```
(tycho) [scox@mac~/dev/tycho/tycho]$ kubectl create -f ../kubernetes/
deployment.extensions/tycho-api created
pod/tycho-api created
clusterrole.rbac.authorization.k8s.io/tycho-api-access created
clusterrolebinding.rbac.authorization.k8s.io/tycho-api-access created
service/tycho-api created
```

Then we use the client as usual.

Usage - C. Within Google Kubernetes Engine from the Google Cloud

Shell

Starting out, Tycho's not running on the cluster:



First deploy the Tycho API

```
$ kubectl create -f ../kubernetes/
deployment.extensions/tycho-api created
pod/tycho-api created
clusterrole.rbac.authorization.k8s.io/tycho-api-access created
clusterrolebinding.rbac.authorization.k8s.io/tycho-api-access created
service/tycho-api created
```

Here we've edited the Tycho service def to create the service as type:LoadBalancer for the purposes of a command line demo. In general, we'll access the service from within the cluster rather than exposing it externally.

That runs Tycho:

☐	tycho-api	OK	Deployment	1/1	default	k8sstagecluster
☐	tycho-api	Running	Pod	1/1	default	k8sstagecluster

Initialize the Tycho API's load balancer IP and node port.

```
$ lb_ip=$(kubectl get svc tycho-api -o json | jq .status.loadBalancer.ingress[0].ip |␣
↪sed -e s,\",,g)
$ tycho_port=$(kubectl get service tycho-api --output json | jq .spec.ports[0].port)
```

Launch an application (deployment, pod, service). Note the `--command` flag is used to specify the command to run in the container. We use this to specify a flag that will cause the notebook to start without prompting for authentication credentials.

```
$ PYTHONPATH=$PWD/.. python client.py --up -n jupyter-data-science-3425 -c jupyter/
↪datascience-notebook -p 8888 --command "start.sh jupyter lab --LabApp.token='
'"
200
{
  "status": "success",
  "result": {
    "containers": {
      "jupyter-data-science-3425-c": {
        "port": 32414
      }
    }
  },
  "message": "Started system jupyter-data-science-3425"
}
```

Refreshing the GKE cluster monitoring UI will now show the service starting:

☐	jupyter-data-science-3425	Creating Service Endpoints	Load balancer	2 / 2	default	k8sstagecluster

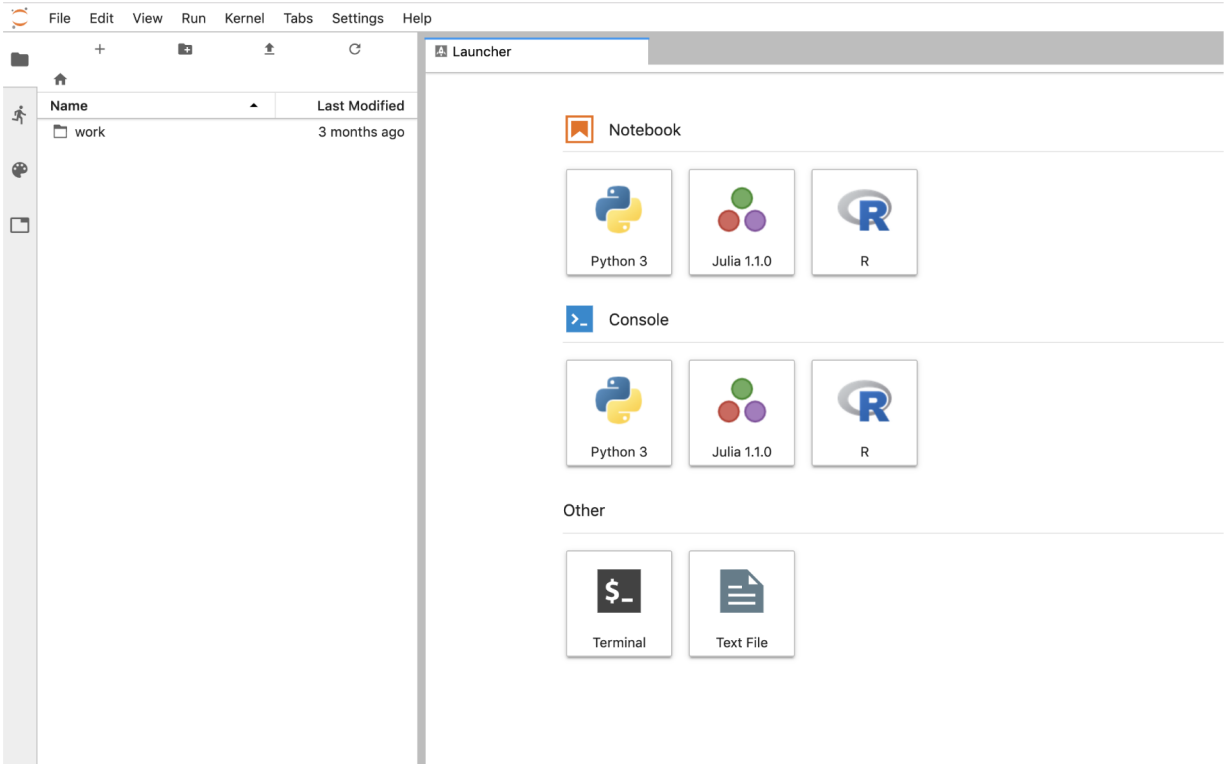
Then running:

☐	jupyter-data-science-3425	Ok	Load balancer	34.67.157.226:8888	1 / 1	default	k8sstagecluster

Get the job's load balancer ip and make a request to test the service.

```
$ job_lb_ip=$(kubectl get svc jupyter-data-science-3425 -o json | jq .status.
↪loadBalancer.ingress[0].ip | sed -e s,\",,g)
$ wget --quiet -O- http://$job_lb_ip:8888 | grep -i /title
    <title>Jupyter Notebook</title>
```

From a browser, that URL takes us directly to the Jupyter Lab IDE:



And shut the service down:

```
$ PYTHONPATH=$PWD/.. python client.py --down -n jupyter-data-science-3425 -s http://
↪$lb_ip:$tycho_port
200
{
  "status": "success",
  "result": null,
  "message": "Deleted system jupyter-data-science-3425"
}
```

This removes the deployment, pod, service, and replicasets created by the launcher.

Client Endpoint Autodiscovery

Using the command lines above without the `-s` flag for server will work on GKE. That is, the client is created by first using the K8s API to locate the Tycho-API endpoint and port. It builds the URL automatically and creates a TychoAPI object ready to use.

```
client_factory = TychoClientFactory ()
client = client_factory.get_client ()
```

6.2 AppStore

The HeLx Appstore is the primary user experience component of the HeLx data science platform. It is a Django based application whose chief responsibilities are authentication and authorization, all visual presentation concerns including transtions between the appstore and apps.

6.2.1 Compute

The system's underlying computational engine is Kubernetes. HeLx runs in a Kubernetes cluster and apps are launched and managed within the same namespace, or administrative context, it uses. Tycho translates the docker compose representation of systems with one ore more containers into a Kubernetes represenation, coordinates and tracks related processes, provides utilization information on running processes, and manages the coordinated deletion of all components when a running application is stopped.

6.2.2 Storage

HeLx apps, in the near term, will mount a read only file system containing reference data and to a writable file system for user data.

6.2.3 Security

HeLx prefers open standard security protocols where available, and applies standards based best practices, especially from NIST and FISMA, in several areas.

6.2.4 Authentication

Authentication is concerned with verifying the identity of a principal and is distinct from determining what actions tha principal is entitled to take in the system. We use OAuth 2.0 facilitated by django allauth framework to integrate Google and GitHub login into the application. In addition SAML based login is also supported for integration with institutional single sign on as needed.

6.2.5 Authorization

Authorization assumes an authenticated principal and is the determination of the actions permitted for that principal. The first layer of authorization is a list of identities allowed access to the system. Email addresses associated with IdP accounts are included in the list. Only principals whose IdPs present an email on the list on their behalf during authentication are authorized to access the system.

6.2.6 Secrets

Data that serves, for example, as a credential for an authentication, must be secret. Since it may not be added to source code control, these values are private to the deployment organization, and must be dynamically injected. This is handled by using Kubernetes secrets during deployment, and trivial keys for developer desktops, all within a uniform process framework. That framework provides a command line interface (CLI) for creating system super users, data migration, starting the application, and packaging executables, among other tasks.

6.2.7 Management CLI

The appstore management CLI provides uniform commands for using the environment. It provides default values for secrets during local development and ways to provide secrets in proudction.

6.2.7.1 Command Description

Command	Description
bin/appstore tests {product}	Run automated unit tests with {product} settings.
bin/appstore run {product}	Run the appstore using {product} settings.
bin/appstore createsuperuser	Create admin user with environment variable provided values.
bin/appstore image build	Build the docker image.
bin/appstore image push	Push the docker image to the repository.
bin/appstore help	Run automated unit tests with {product} settings.

6.2.8 Testing

Automated testing uses the Python standard unittest and Django testing frameworks. Tests should be fast enough to run conveniently, and maximize coverage. For example, the Django testing framework allows for testing URL routes, middleware and other use interface elements in addition to the logic of components.

6.2.9 Packaging

Appstore is packaged as a Docker image. It is a non-root container, meaning the user is not a superuser. It packages a branch of Tycho cloned within the appstore hierarchy.

6.2.10 App Development

HeLx supports metadata driven app development. Apps are expressed using Docker and Docker Compose. AppStore uses the Tycho engine to discover and manage Apps. The Tycho app metadata format specifies the details of each application, contexts to which applications belong, and inheritance relationships between contexts.

App specificatinos are stored in GitHub, each in an application specific subfolder. Along with the docker compose, a .env file specifies environment variables for the application. If a file called icon.png is provided, that is used as the application's icon.

To develop a custom app for AppStore, use the guidelines below:

1) develop one or more docker containers for your app following NIST security best practices
2) fork dockstore-yaml-proposals
3) create a docker-compose for your application and publish it to dockstore-yaml-proposals
4) fork Tycho
5) add your app specific metadata in Tycho registry metadata, including the URL to your application's docker-compose from step 3
6) create pull requests for both Tycho and dockstore-yaml-proposals based off your forks and submit for a code review to us. We'll run your code through our build and security scan pipeline, rejecting any container with high or critical vulnerabilities, subject to further review and either accept or reject the pull requests.

6.2.10.1 Deployment

Appstore is deployed to Kubernetes in production using Helm. The main deployment concerns are: Security: Secrets are added to the container via environment variables. Persistence: Storage must be mounted for a datbaase. Services: The chief dependency is on Tycho which must be at the correct version.

During development, environment variables can be set to control execution:

Variable	Description
DEV_PHASE=[stub, local, dev, val, prod]	In stub, does not require a Tycho service.
ALLOW_DJANGO_LOGIN=[TRUE, FALSE]	When true, presents username and password authentication options.
SECRET_KEY	Key for securing the application.
OAUTH_PROVIDERS	Contains all the providers(google, github).
GOOGLE_CLIENT_ID	Contains the client_id of the provider.
GOOGLE_SECRET	Contains the secret key for provider.
GOOGLE_NAME	Sets the name for the provider.
GOOGLE_KEY	Holds the key value for provider.
GOOGLE_SITES	Contains the sites for the provider.
GITHUB_CLIENT_ID	Contains the client_id of the provider.
GITHUB_SECRET	Contains the secret key of the provider.
GITHUB_NAME	Sets the name for the provider.
GITHUB_KEY	Holds the key value for provider.
GITHUB_SITES	Contains the sites for the provider.
APPSTORE_DJANGO_USERNAME	Holds superuser username credentials.
APPSTORE_DJANGO_PASSWORD	Holds superuser password credentials.
TYCHO_URL	Contains the url of the running tycho host.
OAUTH_DB_DIR	Contains the path for the database directory.
OAUTH_DB_FILE	Contains the path for the database file.
POSTGRES_DB	Contains the connection of the database.
POSTGRES_HOST	Contains the database host.
DATABASE_USER	Contains the database username credentials.
DATABASE_PASSWORD	Contains the database password credentials.
APPSTORE_DEFAULT_FROM_EMAIL	Default email address for appstore.
APPSTORE_DEFAULT_SUPPORT_EMAIL	Default support email for appstore.
ACCOUNT_DEFAULT_HTTP_PROTOCOL	Allows to switch between http and https protocol.

6.2.11 Development Environment

The following script outlines the process:

```
#!/bin/bash

set -ex

# start fresh
rm -rf appstore
#  get a vritualenv
if [ ! -d venv ]; then
    python3 -m venv venv
fi
source venv/bin/activate
# clone appstore
if [ ! -d appstore ]; then
    git clone git@github.com:helxplatform/appstore.git
fi
cd appstore
# use metadata branch and install requirements
git checkout develop
```

(continues on next page)

(continued from previous page)

```
cd appstore
pip install -r requirements.txt

# configure helx product => braini
product=braini
# configure dev mode to stub (run w/o tycho api)
export DEV_PHASE=stub
# create and or migrate the database
bin/appstore updatedb $product
# create the superuser (admin/admin by default)
bin/appstore createsuperuser
# execute automated tests
bin/appstore tests $product
# run the appstore at localhost:8000
bin/appstore run $product
```

6.2.12 Development Environment with Kubernetes

Prerequisites 1. Have Access to a running k8s cluster. 2. Have kubectl set up. 3. Installing kubectl on Linux: 4. Download the latest release 5. Run:

```
curl -LO "https://storage.googleapis.com/kubernetes-release/release/$(curl -s https://
↪storage.googleapis.com/kubernetes-release/release/stable.txt)/bin/
↪linux/amd64/kubectl"
```

6. Make the kubectl binary executable:: chmod +x ./kubectl
7. Move the binary into your PATH:: sudo mn ./kubectl /usr/local/bin/kubectl
8. Check to see if installed:: kubectl version –client

NOTE: Once `kubectl` has been setup then set the `KUBECONFIG` env variable to use other kubeconfigs. For example, the one provided to you will be exported into the terminal where tycho api would be run: export KUBECONFIG=path-to-kubeconfig-file.

Step 1: - Clone the Appstore repo (develop branch):

```
git clone -b develop [https://github.com/helxplatform/appstore.git](https://github.
↪com/helxplatform/appstore.git
```

- Activate virtual environment:

```
python3 -m venv venv

source venv/bin/activate
```

- Install the requirements:

```
pip install -r requirements.txt
```

- Finally run Appstore by using the management CLI:

```
bin/appstore start {product}
```

NOTE: After running bin/appstore start {product} for the first time, please use bin/appstore run {product} every other time. So that migrations to the data-base will only take place once.

Step 2:

- Clone the Tycho repo (develop branch):

```
git clone -b develop [https://github.com/helxplatform/tycho.git](https://github.
↪com/helxplatform/tycho.git)
```

- Activate virtual environment:

```
python3 -m venv venv

source venv/bin/activate
```

- Install the requirements:

```
pip install -r requirements.txt
```

- Export the kubeconfig to the terminal where tycho api is to be run:

```
export KUBECONFIG=path-to-kubeconfig-file
```

- Now run tycho api in the terminal where the kubeconfig was exported:

```
bin/tycho api -d
```

Step 3:

- Now Appstore is running, navigate to the admin panel by appending `/admin` to the url `http://localhost:8000/admin`.
- Login in to the admin panel using `admin/admin` for user/password.
- Nagivate to the application manager: `http://localhost:8000/apps`. From this endpoint we can launch applications.

6.2.12.1 Overview

Architecture

HeLx puts the most advanced analytical scientific models at investigator's finger tips using equally advanced cloud native, container orchestrated, distributed computing systems.

User Experience

Users browse tools of interest and launch those tools to explore and analyze available data. From the user's perspective, HeLx is like an operating system since it runs useful tools. But there's no installation, the data's already present in the cloud with the computation, and analyses can be reproducibly shared with others on the platform.

Compute

The system's underlying computational engine is Kubernetes. HeLx runs in a Kubernetes cluster and apps are launched and managed within the same namespace, or administrative context, it uses. Tycho translates the docker compose representation of systems with one ore more containers into a Kubernetes representation, coordinates and tracks related processes, provides utilization information on running processes, and manages the coordinated deletion of all components when a running application is stopped.

HeLx's cloud native design relies on containerization via Docker for creating reusable modules. These modules are orchestrated using the Kubernetes scheduler. Kubernetes, available at all major cloud vendors and multiple federally funded research platforms including the Texas Advanced Computing Center (TACC) and Pacific Research Platform (PRP) provides an important layer of portability and reproducibility for HeLx.

Within the Kubernetes setting, the system requires a single public IP address. That address routes to a web server acting as a reverse proxy. It partitions the environment into paths requiring an existing authentication and paths that do not. The Appstore interface handles users requests to manage workspaces like Jupyter notebooks and the Nextflow API. For access to applications users have launched, the reverse proxy first consults Appstore to acquire an existing authentication token. If a token exists, the request is forwarded to the programmable reverse proxy which checks the identity of the authenticated user against the path for the launched application. If the user is authorized, the request proceeds.

HeLx's primary user interface is the Appstore, which is a Django based application whose chief responsibilities are authentication and authorization and providing a means to launch apps relevant to each compute platform. Authentication is provided via an OpenID Connect (OIDC) Identity Provider (IdP) such as GitHub or Google as well as a SAML2 IdP. A SAML2 IdP can be configured on a per customer basis using django-saml2-auth module. After a principal is authenticated, AppStore uses a request filter to determine if it is whitelisted. A django middleware intercepts the request and checks the authenticated user's email against an authorized whitelist to grant user access to apps.

Certain data such as credentials for social authentication accounts must not be packaged with source code. Therefore this data must be dynamically injected during image deployment using Kubernetes secrets. AppStore offers a command line interface (CLI) consisting of uniform commands for providing default values for secrets for local deployment and a way to insert them for production deployment. It also provides interfaces for creating system super users, data migration, starting the application, and packaging executables, among other tasks.

AppStore is packaged as a Docker image. It is a non-root container, meaning the user is not a superuser. It packages a branch of Tycho cloned within the AppStore hierarchy.

Launching an application via the AppStore uses the Tycho module which manages a metadata based representation of all available workspaces. Workspaces are defined as docker-compose YAML artifacts. These, in turn, specify sets of cooperating Docker containers constituting a distributed system. Tycho translates these app specifications into Kubernetes constructs to instantiate and run the application. During this process, Tycho also registers the app instance with the Ambassador programmable reverse proxy specifying that only the user that instantiated the application is authorized to access the instance. In addition to configuring the application's networking and authorization interfaces, Tycho also configures the workspace's access to persistent storage. HeLx requires a single Kubernetes construct known as a persistent volume (PV) to designate a storage location. It must be configured as Read-Write-Many to allow multiple Kubernetes deployments to access it simultaneously. Given that configuration, Tycho will configure a sub path on the PV for each user's home directory and mount that path to each container it creates that is a component of an app.

Storage

HeLx apps, in the near term, will mount a read only file system containing reference data and to a writable file system for user data. Storage can also consist of an NFS-rods volume mounted to each container for access to data stored in iRODS.

Security

HeLx prefers open standard security protocols where available, and applies standards based best practices, especially from NIST and FISMA, in several areas.

6.2.12.2 Authentication

Authentication is concerned with verifying the identity of a principal and is distinct from determining what actions the principal is entitled to take in the system. We use the OpenID Connect (OIDC) protocol to federate user identities from an OIDC identity provider (IdP) like Google or GitHub. The OIDC protocol is integrated into the system via open source connectors for the Django environment. This approach entails configuring the application within the platform of each IdP to permit and execute the OIDC handshake.

Authorization

Authentication is provided via an OpenID Connect (OIDC) Identity Provider (IdP) such as GitHub or Google as well as a SAML2 IdP. A SAML2 IdP can be configured on a per customer basis using django-saml2-auth module. After a principal is authenticated, AppStore uses a request filter to determine if it is whitelisted. A django middleware intercepts the request and checks the authenticated user's email against an authorized whitelist to grant user access to apps. Certain data such as credentials for social authentication accounts must not be packaged with source code. Therefore this data must be dynamically injected during image deployment using Kubernetes secrets. AppStore offers a command line interface (CLI) consisting of uniform commands for providing default values for secrets for local deployment and a way to insert them for production deployment. It also provides interfaces for creating system super users, data migration, starting the application, and packaging executables, among other tasks.

Secrets

Data that serves, for example, as a credential for an authentication, must be secret. Since it may not be added to source code control, these values are private to the deployment organization, and must be dynamically injected. This is handled by using Kubernetes secrets during deployment, and trivial keys for developer desktops, all within a uniform process framework. That framework provides a command line interface (CLI) for creating system super users, data migration, starting the application, and packaging executables, among other tasks.

Management CLI

The appstore management CLI provides uniform commands for using the environment. It provides default values for secrets during local development and ways to provide secrets in production.

Command	Description
bin/appstore tests {product}	Run automated unit tests with {product} settings.
bin/appstore run {product}	Run the appstore using {product} settings.
bin/appstore createsuperuser	Create admin user with environment variable provided values.
bin/appstore image build	Build the docker image.
bin/appstore image push	Push the docker image to the repository.
bin/appstore image run {product}	Run automated unit tests with {product} settings.
bin/appstore help	Run automated unit tests with {product} settings.

Testing

Automated testing uses the Python standard unittest and Django testing frameworks. Tests should be fast enough to run conveniently, and maximize coverage. For example, the Django testing framework allows for testing URL routes, middleware and other use interface elements in addition to the logic of components.

Packaging

Appstore is packaged as a Docker image. It is a non-root container, meaning the user is not a superuser. It packages a branch of Tycho cloned within the appstore hierarchy.

Launching an application via the AppStore uses the Tycho module which manages a metadata based representation of all available workspaces. Workspaces are defined as docker-compose YAML artifacts. These, in turn, specify sets of cooperating Docker containers constituting a distributed system. Tycho translates these app specifications into Kubernetes constructs to instantiate and run the application. During this process, Tycho also registers the app instance with the Ambassador programmable reverse proxy specifying that only the user that instantiated the application is authorized to access the instance. In addition to configuring the application's networking and authorization interfaces, Tycho also configures the workspace's access to persistent storage. HeLx requires a single Kubernetes construct known as a persistent volume (PV) to designate a storage location. It must be configured as Read-Write-Many to allow multiple Kubernetes deployments to access it simultaneously. Given that configuration, Tycho will configure a sub path on the PV for each user's home directory and mount that path to each container it creates that is a component of an app.

App Development

During development, environment variables can be set to control execution:

Variable	Description
DEV_PHASE=[stub, local, dev, val, prod	In stub, does not require a Tycho service.
ALLOW_DJANGO_LOGIN=[TRUE, FALSE]	When true, presents username and password authentication options.
SECRET_KEY	Key for securing the application.
OAUTH_PROVIDERS	Contains all the providers(google, github).
GOOGLE_CLIENT_ID	Contains the client_id of the provider.
GOOGLE_SECRET	Contains the secret key for provider.
GOOGLE_NAME	Sets the name for the provider.
GOOGLE_KEY	Holds the key value for provider.
GOOGLE_SITES	Contains the sites for the provider.
GITHUB_CLIENT_ID	Contains the client_id of the provider.
GITHUB_SECRET	Contains the secret key of the provider.
GITHUB_NAME	Sets the name for the provider.
GITHUB_KEY	Holds the key value for provider.
GITHUB_SITES	Contains the sites for the provider.
APPSTORE_DJANGO_USERNAME	Holds superuser username credentials.
APPSTORE_DJANGO_PASSWORD	Holds superuser password credentials.
TYCHO_URL	Contains the url of the running tycho host.
OAUTH_DB_DIR	Contains the path for the database directory.
OAUTH_DB_FILE	Contains the path for the database file.
POSTGRES_DB	Contains the connection of the database.
POSTGRES_HOST	Contains the database host.
DATABASE_USER	Contains the database username credentials.
DATABASE_PASSWORD	Contains the database password credentials.
APPSTORE_DEFAULT_FROM_EMAIL	Default email address for appstore.
APPSTORE_DEFAULT_SUPPORT_EMAIL	Default support email for appstore.
ACCOUNT_DEFAULT_HTTP_PROTOCOL	Allows to switch between http and https protocol.

App Metadata

Making application development easy is key to bringing the widest range of useful tools to the platform so we prefer metadata to code wherever possible for creating HeLx Apps. Apps are systems of cooperating processes. These are expressed using Docker and Docker Compose. Appstore uses the Tycho engine to discover and manage Apps. The Tycho app metadata format specifies the details of each application, contexts to which applications belong, and inheritance relationships between contexts.

Docker compose syntax is used to express cooperating containers comprising an application. The specifications are stored in GitHub, each in an application specific subfolder. Along with the docker compose, a `.env` file specifies environment variables for the application. If a file called icon.png is provided, that is used as the application's icon.

Development Environment

More information coming soon. The following script outlines the process:

```
#!/bin/bash

set -ex

# start fresh
rm -rf appstore
#  get a vritualenv
if [ ! -d venv ]; then
    python3 -m venv venv
fi
source venv/bin/activate
# clone appstore
if [ ! -d appstore ]; then
    git clone git@github.com:helxplatform/appstore.git
fi
cd appstore
# use metadata branch and install requirements
git checkout metadata
cd appstore
pip install -r requirements.txt

# configure helx product => braini
product=braini
# configure dev mode to stub (run w/o tycho api)
export DEV_PHASE=stub
# create and or migrate the database
bin/appstore updatedb $product
# create the superuser (admin/admin by default)
bin/appstore createsuperuser
# execute automated tests
bin/appstore tests $product
# run the appstore at localhost:8000
bin/appstore run $product
```

6.2.12.3 Applications

Appstore is deployed to Kubernetes in production using Helm. The main deployment concerns are:

- **Security** : Secrets are added to the container via environment variables.

- **Persistence** : Storage must be mounted for a database.
- **Services** : The chief dependency is on Tycho which must be at the correct version.

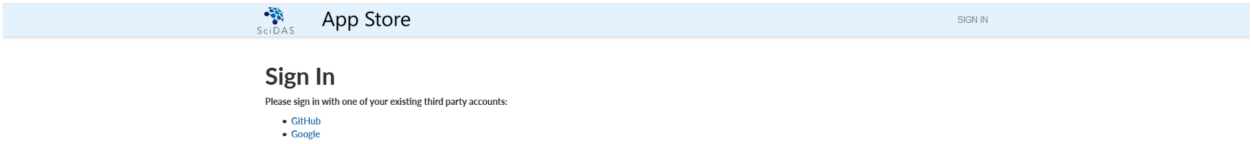
The Helm chart for deploying Appstore can be found here.

Creating an Application

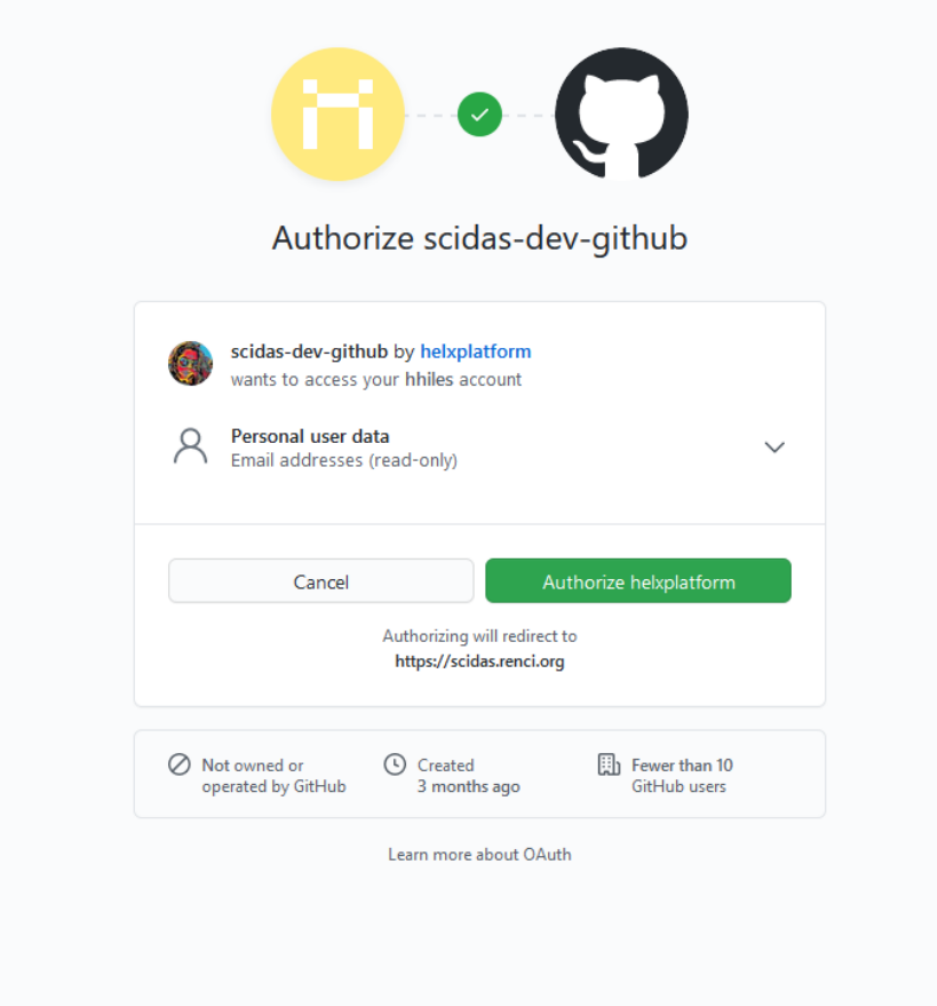
The instance of HeLx you start with will vary. This tutorial shows starting an app from the SciDAS instance, but instructions are the same regardless of how you want to start an App using HeLx.

Log in and Start your App

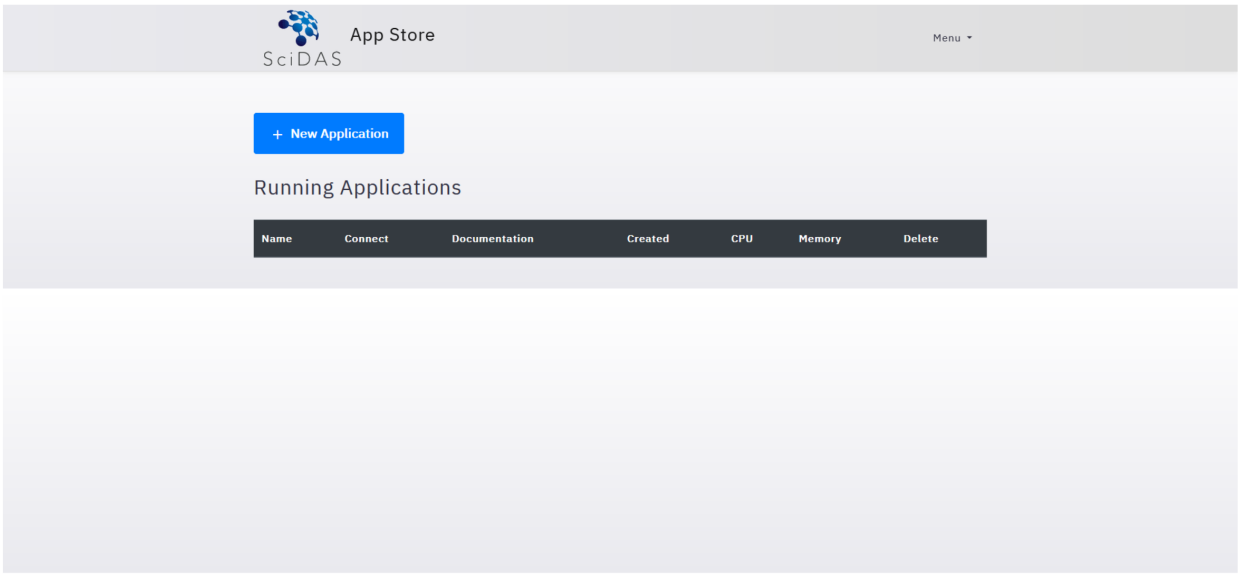
Step-1: Begin by navigating to your team's instance of HeLx and signing in using either GitHub or Google.



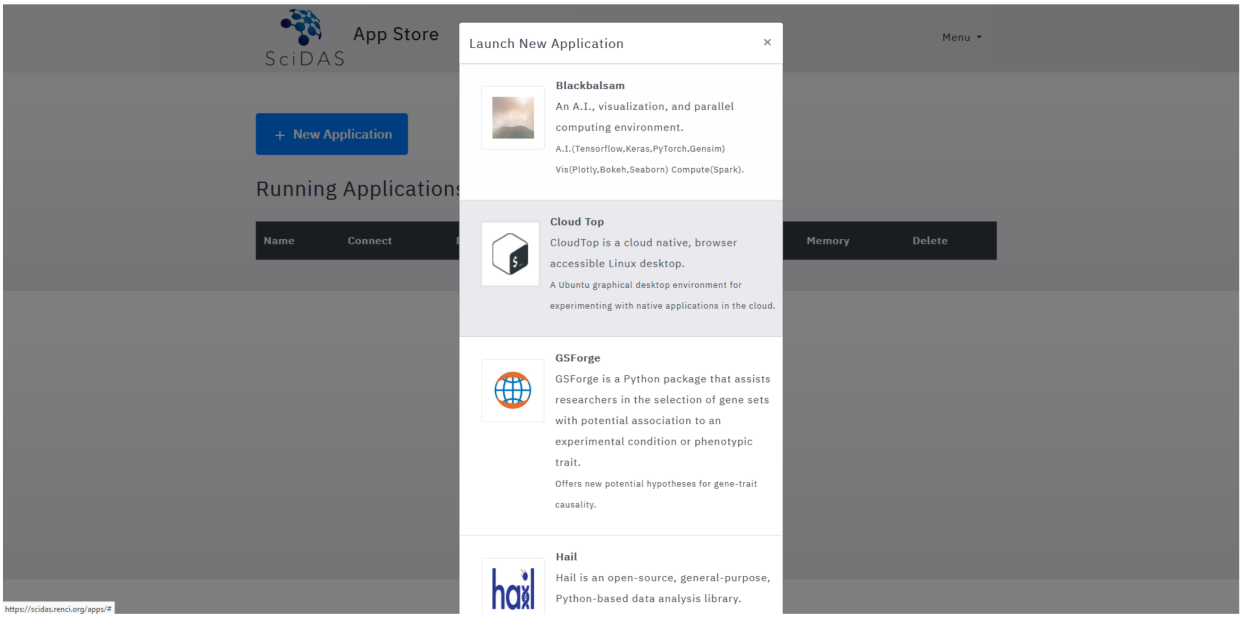
Step-2: Log in to HeLx.



Step-3: Select "New Application" to choose the App you wish to run.



Then



Step-4: Once you have launched your App, use the following guides for instructions on launching the following Existing Apps:

- Blackbalsam,
- CloudTop,
- DICOM Viewer for Google Health API,
- ImageJ-Napari,
- Jupyter-DataScience,
- Nextflow API, and
- Rstudio

Blackbalsam

Begin by starting the App as described in the section Creating an Application. Select the CloudTop Viewer application.

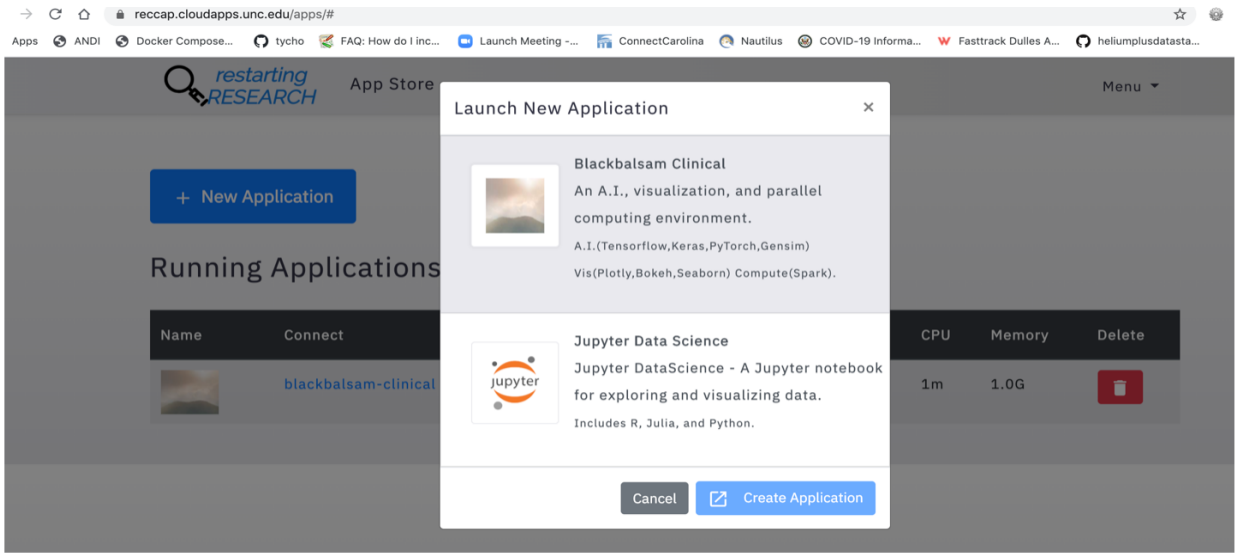
Exporting REDCap Data into R

Step-1: Get a REDCap account for a project URL and API key

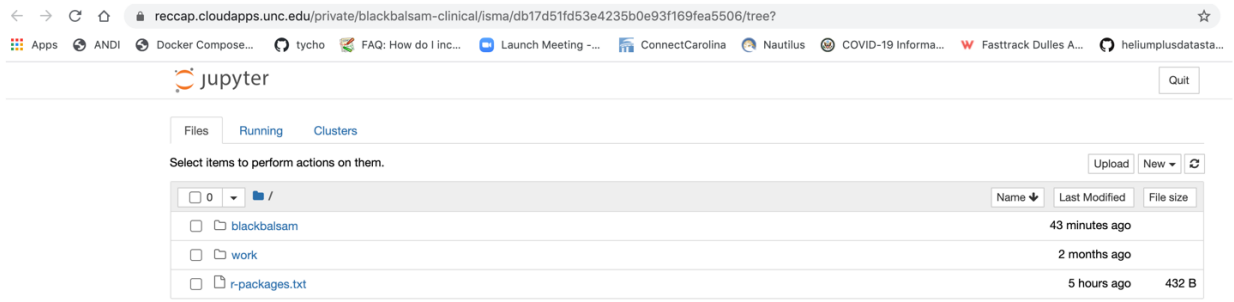
if you don't have an account contact Adam Lee to get one. If you have an existing account use the credentials associated with it

Step-2: Go to https://reccap.cloudapps.unc.edu/accounts/login

Step-3: Click "New Application" and choose **Blackbalsam Clinical**



Step-4: Click on "black balsam" folder and then click on RedCapAPIRDemo.ipynb



Step-5: In the notebook, on cell 2 substitute REDCap URL and API key with your credentials

Establishing a connection to the redcap project.

In [4]:
```
RedCap_URL <- "<put-project-url-here>"
API_KEY <- "<Put-api-token--here>"
rcon <- redcapConnection(url=RedCap_URL, token=API_KEY)
```

Step-6: Run first three cells in the notebook

Dependencies redcapAPI library, API_KEY, and project url.

In [1]:
```
# Importing the library
library(redcapAPI)
```

Establishing a connection to the redcap project.

In [4]:
```
RedCap_URL <- "<put-project-url-here>"
API_KEY <- "<Put-api-token--here>"
rcon <- redcapConnection(url=RedCap_URL, token=API_KEY)
```

Exporting project metadata from redcap.

The metadata consists of various keys like field_type, form_name, field_name, identifier etc.

In [5]:
```
export_metadata <- exportMetaData(rcon,
                  fields = NULL,
                  forms = NULL,
                    )
```

Step-7: To export records, use show_forms, show_fields, or show_all_records cells

Exporting Records by specifying different parameters such as records, fields,and forms.

In [ ]:
```
# This is how a specific field can be exported by providing its colClass.
show_fields =exportRecords(
  rcon,
  fields = 'eligibility_1',
  records = NULL,
  colClasses = 'eligibility_questions'
    )

show_fields
```

In [ ]:
```
# This is how a specific form can be exported by specifying the form name in the form paremeter.
show_forms =exportRecords(
    rcon,
    forms = "consent_to_participate",
    records = NULL,
    )
show_forms
```

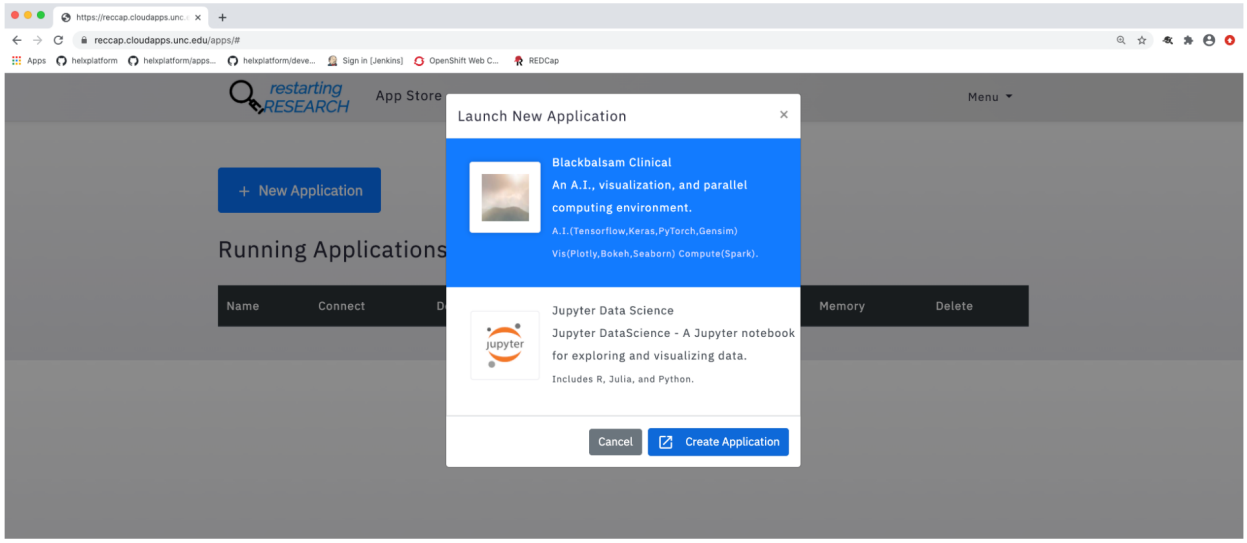
In [ ]:
```
# By assigning NULL value to the exportRecords function parameters all of the records can be exported at once.
show_all_records =exportRecords(
    rcon,
    forms = NULL,
    records = NULL,
)

show_all_records
```

Using the restartr notebook to get data into mongodb

Step-1: Go to https://reccap.cloudapps.unc.edu/accounts/login and login.

Step-2: Click on "New Application", choose Blackbalsam Clinical then click "Create Application".



Step-3: Click on the "blackbalsam" directory and then launch the RestartrApiDemo.ipynb notebook.

Step-4: In cells 3 to 6 substitute your api-key here "".

Example to call the observation api for restartr below:

In [ ]:
```
x = requests.post('https://reccap.cloudapps.unc.edu/api/observation',
                  headers={"Content-Type": "application/json", "X-API-Key": "<your api key>"},
                  data=r)
print("The result of observation api", x.text)
```

Example to call the query api for restartr below:

In [ ]:
```
y = requests.post('https://reccap.cloudapps.unc.edu/api/query',
                  headers={"Content-Type": "application/json", "X-API-Key": "<your api key>"},
                  data=r)
print("The result of query api", y.text)
```

Example to query records by "_id" below:

In [ ]:
```
# To query records by "_id" substitute the id below <place-id-here> after calling the observation api
# which will return an id .
# Example: s = {"_id": "5f2c507e8b501271fb7b1d76"}.
# The id above in the example can also be uses to query a record that is already present in the database.

s = {"_id": "<put-id-here>"}
y = requests.post('https://reccap.cloudapps.unc.edu/api/query',
                  headers={"Content-Type": "application/json", "X-API-Key": "<put-api-key-here>"},
                  data=json.dumps(s))
print( y.text)
```

Example to query records by sub-field below:

In [ ]:
```
# To query by sub-field the data will have to be formated as it is below.
# To run a query on existing data in database just use the Example below.
# Using the EXAMPLE below will return all data that has a StudyId of 100160.
# To run a query on a recently sent observation modify job_1 accordingly.

"""EXAMPLE:
job = {
    "results.StudyId" : 100160,
    "byField": "results.StudyId",
} """

job_1 = {
    "results.<put-sub-field-here>" : <put-sub-field-value>,
    "byField": "results.<put-sub-field-here>",
}
i = json.dumps(job_1)
y = requests.post('https://reccap.cloudapps.unc.edu/api/query',
                  headers={"Content-Type": "application/json", "X-API-Key": "<put-api-key-here>"},
                  data=i)
print("The result of query api", y.text)
```

Step-5: Run the first two cells in the notebook.

This code is a demo for calling the Restartr API.

Importing the required libraries below.

In [12]:
```
import requests
import json
```

Format your data below:

In [ ]:
```
job = {
    "kind" : "test",
    "value" : "10"
}
r = json.dumps(job)
```

Step-6: Run the third cell to persist data into mongodb.

Example to call the observation api for restartr below:

In [ ]:
```
x = requests.post('https://reccap.cloudapps.unc.edu/api/observation',
                  headers={"Content-Type": "application/json", "X-API-Key": "<your api key>"},
                  data=r)
print("The result of observation api", x.text)
```

- The first cell imports the required dependencies.
- The second cell is where data can be formatted to insert into mongodb.

Step-7: To query the data use, query by "_id", query by sub-field, or just use query methods provided in the RestartrApiDemo.ipynb notebook.

Example to call the query api for restartr below:

In [ ]:
```
y = requests.post('https://reccap.cloudapps.unc.edu/api/query',
                  headers={"Content-Type": "application/json", "X-API-Key": "<your api key>"},
                  data=r)
print("The result of query api", y.text)
```

Example to query records by "_id" below:

In [ ]:
```
# To query records by "_id" substitute the id below <place-id-here> after calling the observation api
# which will return an id .
# Example: s = {"_id": "5f2c507e8b501271fb7b1d76"}.
# The id above in the example can also be uses to query a record that is already present in the database.

s = {"_id": "<put-id-here>"}
y = requests.post('https://reccap.cloudapps.unc.edu/api/query',
                  headers={"Content-Type": "application/json", "X-API-Key": "<put-api-key-here>"},
                  data=json.dumps(s))
print( y.text)
```

Example to query records by sub-field below:

In [ ]:
```
# To query by sub-field the data will have to be formated as it is below.
# To run a query on existing data in database just use the Example below.
# Using the EXAMPLE below will return all data that has a StudyId of 100160.
# To run a query on a recently sent observation modify job_1 accordingly.

"""EXAMPLE:
job = {
    "results.StudyId" : 100160,
    "byField": "results.StudyId",
} """

job_1 = {
    "results.<put-sub-field-here>" : <put-sub-field-value>,
    "byField": "results.<put-sub-field-here>",
}
i = json.dumps(job_1)
y = requests.post('https://reccap.cloudapps.unc.edu/api/query',
                  headers={"Content-Type": "application/json", "X-API-Key": "<put-api-key-here>"},
                  data=i)
print("The result of query api", y.text)
```

CloudTop

On this page you will find guides for testing the CloudTop Docker, the CloudTop OHIF Docker, and the CloudTop ImageJ/Napari Docker.

Testing the CloudTop Docker

Begin by starting the App as described in the section Creating an Application. Select the CloudTop Viewer application.

Step 1: Run the CloudTop Docker

```
docker run -p8080:8080 -e USER_NAME=howard -e VNC_PW=test heliumdatastage/
↪cloudtop:latest
```

`USER_NAME` and `VNC_PW` can be whatever you want: those are the authentication info you will need to log in for Step 2. Change the tag to whichever tag you want to test.

Step 2: Connect to the running docker

1. Browse to `localhost:8080`
2. **Enter the `USERNAME` and `VNC_PW` you specified when starting the** `Dockerfile`
3. Press the Login button
4. Wait for Guacamole to respond

Step 3: Make sure the home directory is OK

1. Start a terminal emulator from the applications menu. In the resultant shell, type

```
echo $HOME
```

1. You should see `/home/USER_NAME` where `USER_NAME` is the user name specified in
2. Note the presence of the Firefox browser icon

At this point the basic CloudTop functionality is working.

Testing the CloudTop OHIF Docker

Step 1: Run the CloudTop OHIF docker

```
docker run -p3000:3000 -p8080:8080 -e
"CLIENT_ID=OUR_CLIENT_ID.apps.googleusercontent.com" -e
USER_NAME=howard -e VNC_PW=test heliumdatastage/cloudtop-
ohif:latest
```

where `OUR_CLIENT_ID` is found in the file.

`google_health.env` file in the renci_data_stage directory of our keybase account. `USER_NAME` and `VNC_PW` can be whatever you want: those are the authentication info you will need to log in for Step 2. Change the tag to whichever tag you want to test.

Step 2: Connect to the running docker

1. Browse to `localhost:8080`
2. **Enter the `USERNAME` and `VNC_PW` you specified when starting the** Dockerfile
3. Press the Login button - Wait for Guacamole to respond

Step 3: Make sure the home directory is OK

1. Start a terminal emulator from the applications menu. In the resultant shell, type:

```
echo $HOME
```

2. You should see `/home/USER_NAME` where `USER_NAME` is the user name specified in Step 1
3. Note the presence of the Firefox browser icon

At this point the basic CloudTop functionality is working.

Step 4: Test the OHIF functionality

1. Exit the terminal emulator by typing `exit`
2. Click the Firefox icon and browse to `localhost:3000`
3. At this point you will be prompted for your Google user ID.
4. Click Next.

5. Google may prompt you to choose the account you wish to proceed with. If prompted, pick your G Suite account.
6. Click Next.
7. Enter your password. The browser will ask if you want to save the password. It doesn't matter if you do or not
8. Respond to the 2 step authentication. If you haven't used it before, you may be prompted to set up the 2 step authentication.
9. You should now see the basic OHIF screen with a large selection of projects.

Step 5: Browse to Your Data Set

1. Select helx-dev
2. Select the northamerica- northeast1 region
3. Select the DicomTestData dataset
4. Select the TestData Dicom Store
5. You should now see our test datasets.

Chose your test data set and have fun!

Testing the CloudTop ImageJ/Napari Docker

Step 1: Start the Docker

1. Start the docker with the following command:

```
docker run -p8080:8080 -e USER_NAME=howard -e VNC_PW=test
heliumdatastage/cloudtop-image-napari:latest
```

where `USER_NAME` and `VNC_PW` can be whatever you want: those are the authentication info you will need to log in for Step 2. Change the tag to whichever tag you want to test.

Step 2: Connect to the running docker

1. Browse to `localhost:8080`
2. **Enter the `USERNAME` and `VNC_PW` you specified when starting the** Dockerfile
3. Press the Login button
4. Wait for Guacamole to respond

Step 3: Make sure the home directory is OK

1. Start a terminal emulator from the applications menu. In the resultant shell, type:

```
echo $HOME
```

2. You should see `/home/USER_NAME` where `USER_NAME` is the user name specified in Step 1
3. Note the presence of the ImageJ, Napari and Firefox browser icon. **If any are missing the test fails.**
4. At this point the basic CloudTop functionality is working. Next we will want to verify that ImageJ and Napari are working

Step 4: Make sure the ImageJ application launcher works correctly

1. Exit the terminal application and click the ImageJ icon. There is no ImageJ test data included in the docker.
2. Exit ImageJ and make sure the Napari application launcher works correctly.

3. The docker does not contain any test data. The docker test is now complete.
4. Exit Napari and stop the docker.

DICOM Viewer for Google Health API

Running the DICOM Viewer Application

Begin by starting the App as described in the section Creating an Application. Select the CloudTop Viewer application.

Step 1: Connect to the application

- Enter the `USERNAME` and `VNC_PW` you have been provided
- Press the Login button
- Wait for the app to respond

Step 2: Use OHIF functionality

- Click the Firefox icon and browse to `localhost:3000`
- At this point you will be prompted for your Google user ID.
- Click Next.
- Google may prompt you to choose the account you wish to proceed with. If prompted, use the account with which you logged in to the App Store
- Click Next.
- Enter your password. The browser will ask if you want to save the password. It doesn't matter if you do or not
- Respond to the 2 step authentication. If you haven't used it before, you may be prompted to set up the 2 step authentication.
- You should now see the basic OHIF screen with a large selection of projects.

Step 3: Browse to The example Data Set

- Select helx-dev
- Select the northamerica- northeast1 region
- Select the DicomTestData dataset
- Select the TestData Dicom Store
- You should now see our test datasets.

Chose your test data set and have fun!

ImageJ-Napari

Coming soon.

Jupyter-DataScience

Begin by starting the App as described in the section Creating an Application. Select the CloudTop Viewer application.

Introduction

Jupyter/datascience includes popular packages for data analysis from the Python, Julia and R communities and also packages are included from its ancestor images **jupyter/sci-py notebook** , **jupyter/r-notebook** and **jupyter/minimal-notebook**.

Some of the packages it includes are:

dask, pandas, numexpr, matplotlib, scipy, seaborn, scikit-learn, scikit-image, sympy, cython, patsy, statsmodel, cloudpickle, dill, numba, bokeh, sqlalchemy, hdf5, vincent, beautifulsoup, protobuf, xlrd, bottleneck, and pytables packages .

ipywidgets and ipymp for interactive visualizations and plots in Python notebooks.

Facets for visualizing machine learning datasets.

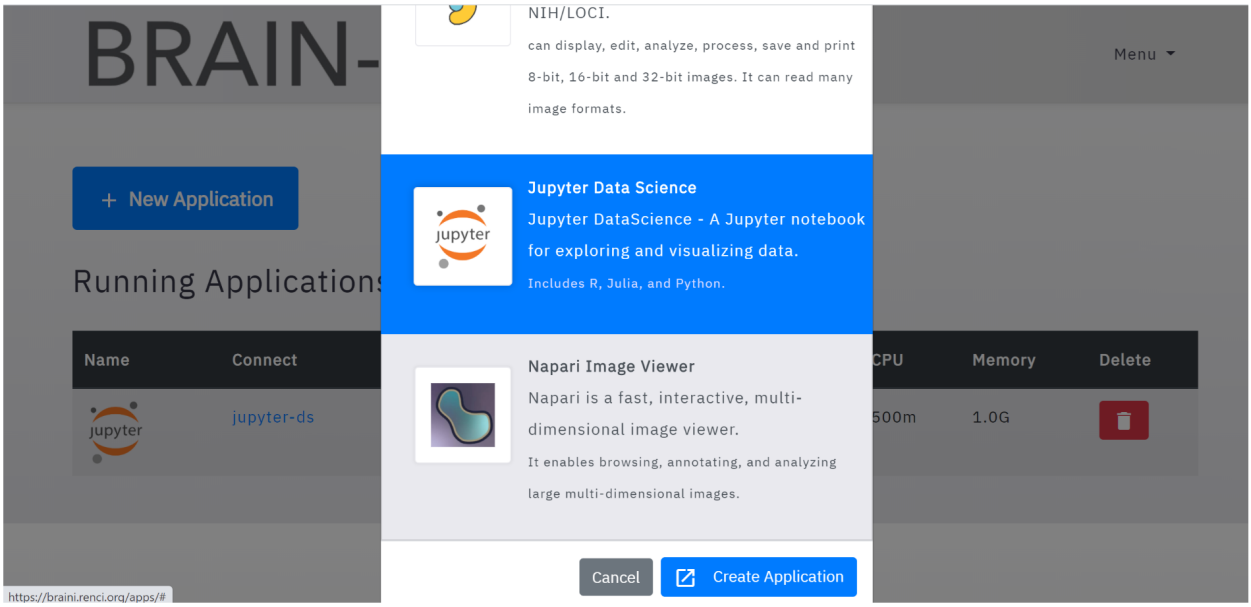
The Julia compiler and base environment.

IJulia to support Julia code in Jupyter notebooks.

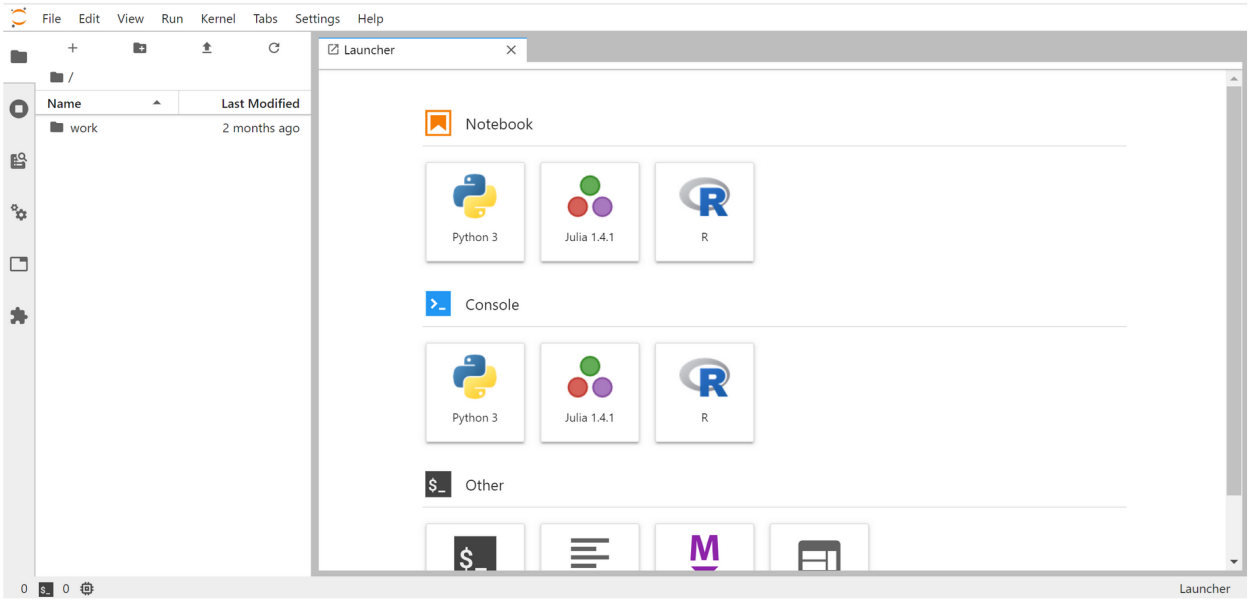
HDF5, Gadfly, and RDatasets packages.

Working with jupyter-datascience notebook in HeLx

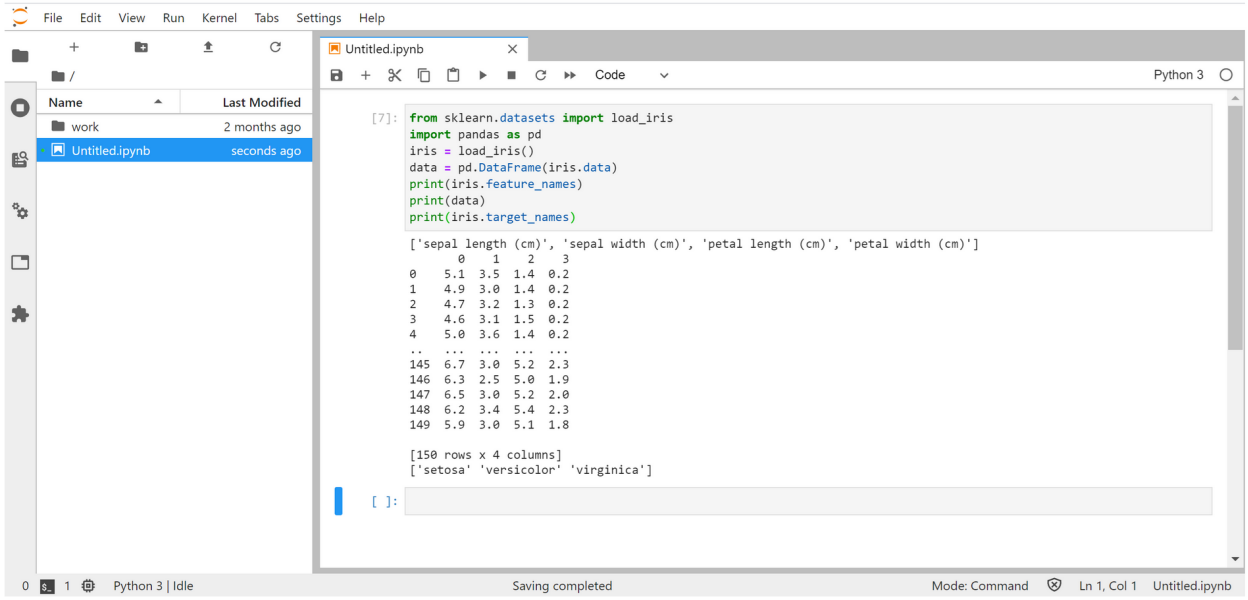
Step-1: Launch a jupyter-datascience notebook from HeLx by clicking on "Launch Application" button.



Step-2: This brings us to the jupyter-lab panel where we can select the environment that we want to work on (Python, Julia, R).



Step-3: Start working on it. Below code shows loading iris dataset (features, labels) from sklearn package to train/test our machine learning model.



Nextflow API

Begin by starting the App as described in the section Creating an Application. Select the Nextflow API application.

Introduction

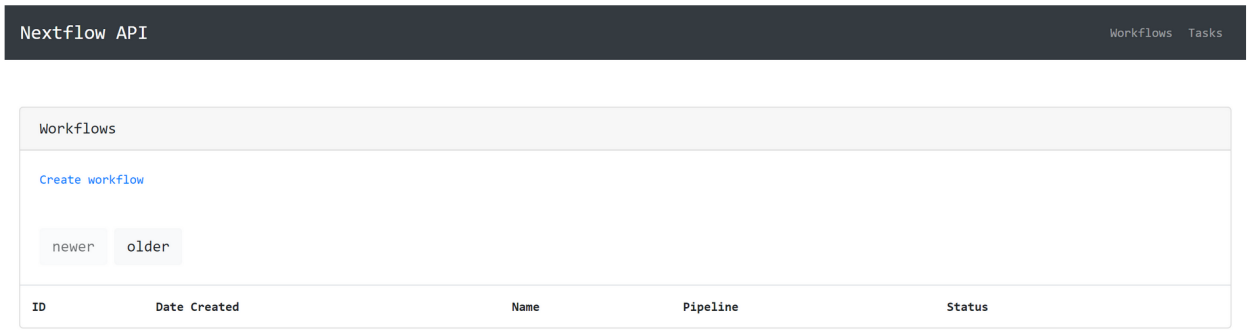
Nextflow enables scalable and reproducible scientific workflows using software containers. It allows the adaptation of pipelines written in the most common scripting languages.

Its fluent DSL simplifies the implementation and the deployment of complex parallel and reactive workflows on clouds and clusters.

Working with Nextflow on HeLx

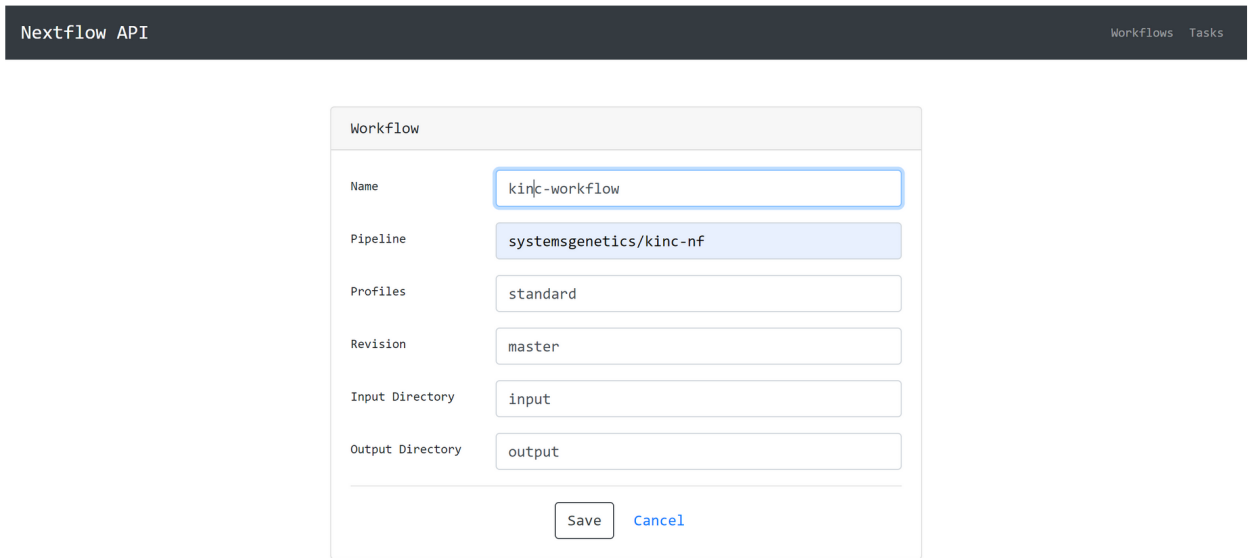
Step-1:

- After selecting the Nextflow API application, you will be taken to the Nextflow API home page, where we can view the launched workflows and create new workflows.



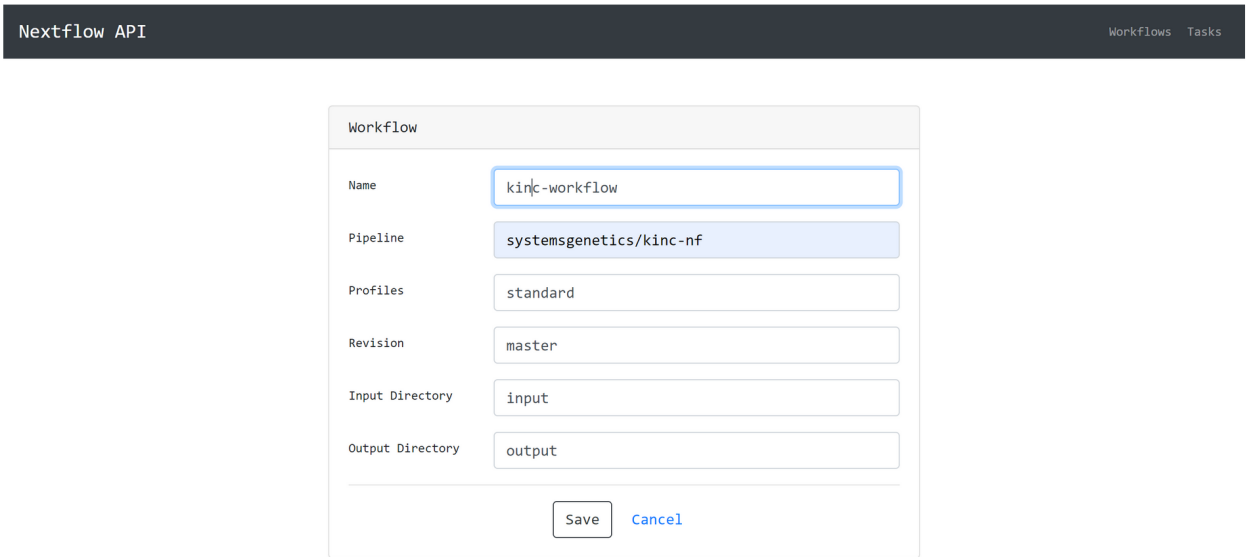
Step-2:

- Below is a demo of how to launch a systemsgenetics/kinc workflow.
- Click on "Create Workflow" button and fill in the form to give it a "Name" and specify the Pipeline (in this case *systemsgenetics/kinc-nf*).



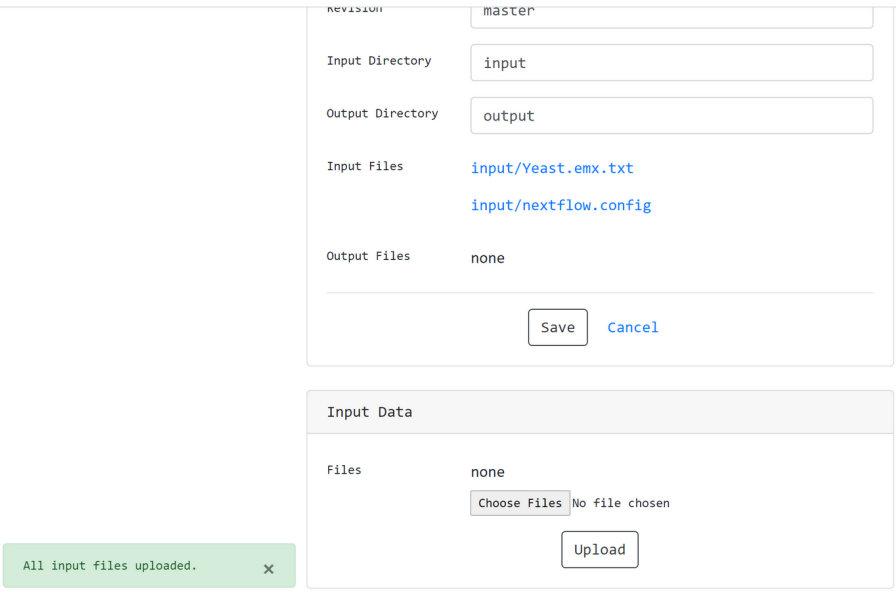
Step-3:

- Uploading the necessary files, a GEM file in the format "*.emx.txt" and a nextflow.config file (can upload all files at once).
- Click on "Upload" button.



Step-4:

- Now we are all set to launch the workflow. Go ahead and click on “Launch” button.
- This should show all the logs of the processes/jobs running in the background on the Kubernetes cluster.



RStudio

Coming soon.

Indices and tables

- genindex
- modindex

- search
- genindex
- modindex
- search

6.3 Installation

6.3.1 Installing HeLx

6.3.1.1 Prerequisites

1. Install GCloud sdk and configure for your project
2. Install Helm3
3. Install Git

Optional Fourth Step: Set up GitHub or Google OAuth credentials if configuring social auth for your install

GitHub

1. In your GitHub account, go to Settings->Developer Settings
2. On the left panel, select OAuth Apps -> New OAuth App
3. Enter the application name i.e `helx-github`
4. Set Homepage URL -> `https://[your hostname]/accounts/login`
5. Set Authorization Callback URL -> `https://[your hostname]/accounts/github/login/callback/`
6. Record the values for `GITHUB_NAME`, `GITHUB_CLIENT_ID`, and `GITHUB_SECRET` to be used in deployment later

Google

1. Log in to your GCP account and navigate to API & Services->Credentials
2. Create a new OAuth client ID with the application type of Web application
3. Set Authorized JavaScript origins URIs -> `https://[your hostname]`
4. Set Authorized redirect URIs -> to `https://[your hostname]/accounts/google/login/callback/`
5. After the credentials are created record `GOOGLE_NAME`, `GOOGLE_CLIENT_ID`, and `GOOGLE_SECRET` to be used in deployment later

6.3.1.2 GKE Install using Helm

Access to a GKE Cluster

Obtain access to a GKE cluster (For instructions on setting up a cluster, refer to this Quickstart)

Deployment

Watch a video of this process here.

1. Use the helm repo command to add HeLx chart repository

```
helm repo add helxplatform https://helxplatform.github.io/devops/charts
```

This will add the repo into your helm repositories list if it doesn't already exist.

2. Install HeLx using the following command:

```
helm -n helx install helx helxplatform/helx --values helx-pjl-values.yaml
```

Following is some output from the helm install command:

```
[vagrant@localhost helxplatform]$ helm install helx devops/helx
NAME: helx
LAST DEPLOYED: Tue Nov 17 21:40:55 2020
NAMESPACE: default
STATUS: deployed
REVISION: 1
NOTES:
Use the following commands to get information about how to log in to the HeLx website.
DJANGO_ADMIN_USERNAME=`kubectl get secret csappstore-secret -o jsonpath="{.data.
↪APPSTORE_DJANGO_USERNAME}" | base64 --decode`
DJANGO_ADMIN_PASSWORD=`kubectl get secret csappstore-secret -o jsonpath="{.data.
↪APPSTORE_DJANGO_PASSWORD}" | base64 --decode`
HELX_IP=`kubectl get svc nginx-revproxy -o jsonpath="{.status.loadBalancer.ingress[*].
↪ip}"`
printf "Django admin username: $DJANGO_ADMIN_USERNAME\nDjango admin password: $DJANGO_
↪ADMIN_PASSWORD\nHeLx URL: http://$HELX_IP\nDjango admin URL: http://$HELX_IP/admin\n
↪"
```

Example:

```
[vagrant@localhost helxplatform]$ DJANGO_ADMIN_USERNAME=`kubectl get secret␣
↪csappstore-secret -o jsonpath="{.data.APPSTORE_DJANGO_USERNAME}" | base64 --decode`
[vagrant@localhost helxplatform]$ DJANGO_ADMIN_PASSWORD=`kubectl get secret␣
↪csappstore-secret -o jsonpath="{.data.APPSTORE_DJANGO_PASSWORD}" | base64 --decode`
[vagrant@localhost helxplatform]$ HELX_IP=`kubectl get svc nginx-revproxy -o jsonpath=
↪"{.status.loadBalancer.ingress[*].ip}"`
[vagrant@localhost helxplatform]$ printf "Django admin username: $DJANGO_ADMIN_
↪USERNAME\nDjango admin password: $DJANGO_ADMIN_PASSWORD\nHeLx URL: http://$HELX_
↪IP\nDjango admin URL: http://$HELX_IP/admin\n"
Django admin username: admin
Django admin password: sHoc6OqUNIhRHs1YHhtF
HeLx URL: http://34.73.96.240
Django admin URL: http://34.73.96.240/admin
```

An admin username and password is created as part of the install process to allow access to the application and configure other users and social auth.

To access HeLx admin portal, use the admin URL shown above for your install i.e. `http://34.73.96.240/admin`.

To launch apps, navigate to apps URL for your install i.e. `http://34.73.96.240/apps`.

Although an admin user has the ability to launch apps, ideally, a non-admin HeLx user should be added to Django Users and Authorized Users on the admin portal.

Adding a user and OAuth credentials in Django admin portal

1. Add a new user for your HeLx instance Authentication and Authorization->Users->Add
2. Whitelist the newly created user Core->Authorized Users->Add
3. Set up social account(s) in Django admin Social Accounts->Social Applications->Add

   NOTE: Use the Client ID and token from GitHub/Google OAuth setup in prerequisites

Cleanup

To delete HeLx run this command:

```
helm delete helx
```

NOTE: You will need to delete any apps created with HeLx using the web UI or manually with kubectl commands.

6.3.1.3 Standard K8S Install Using a HeLx Install Script

Prerequisites

These instructions assume you have cluster admin permissions for your cluster.

1. Set up a loadbalancer (we use MetalLB) for the cluster
2. Set up an NFS server for persistent storage
3. Create a namespace kubectl create namespace `<<helx-username>>`
4. Create two NFS subdirectories based on namespace `/srv/k8s-pvs/namespace/appstore /srv/k8s-pvs/namespace/stdnfs`
5. Allocate an IP address for your helx website i.e. `192.168.0.1`
6. Create a DNS record for your helx website i.e. helx.example.com
7. Create a TLS certificate for your website and a secret for the certificate

PV and PVC for appstore and stdnfs will be created as part of the deployment script later.

Access to install script

1. Use Git to clone the devops repo using the following command: `git clone https://github.com/helxplatform/devops.git`
2. Navigate to devops/bin and copy `env-vars-template.sh` to an env specific properties file for your cluster `cp env-vars-template.sh env-vars-clustername-helx.sh`
3. Edit the env vars file to be more specific to the cluster env you have set up earlier.

4. Add variables for `GITHUB_NAME`, `GITHUB_CLIENT_ID`, and `GITHUB_SECRET` in the variables file and assign the corresponding values after the OAuth App is created.

Deploy

To deploy tycho, ambassador, nginx, and appstore, run the following:

```
./k8s-apps.sh -c env-vars-blackbalsam-igilani-helx.sh deploy all
```

To deploy specific components such as tycho run:

```
./k8s-apps.sh -c env-vars-blackbalsam-igilani-helx.sh deploy tycho
```

Cleanup

To delete all deployments:

```
./k8s-apps.sh -c env-vars-blackbalsam-igilani-helx.sh delete apps
```

Please note that PVs/PVCs will need to be deleted separately. To delete everything including the PVs and PVCs, you can use:

```
./k8s-apps.sh -c env-vars-blackbalsam-igilani-helx.sh delete all
```

To delete a specific deployment:

```
./k8s-apps.sh -c env-vars-blackbalsam-igilani-helx.sh tycho
```

DevOps

To deploy HeLx 1.0 on the Google Kubernetes Engine you will need to have an account with Google Cloud Platform and configure the ‘Google Cloud SDK on your local system.

Check your Google Cloud SDK is setup correctly.

```
glcoud info
```

Decide which directory you want the code to deploy HeLx to be and execute the following commands to checkout code from their GitHub repositories. Some variables will need to be changed. These commands were done in a BASH shell checked on MacOS and will probably work on Linux, maybe on Windows if you use Cygwin, the Windows Subsystem for Linux (WSL), or something similar. Most variables can be set as either environment variables or within the configuration file. Look towards the top of “devops/bin/gke-cluster.sh” to see a list of variables.

```
# Set the Google project ID that you want the cluster to be created in.
PROJECT="A_GOOGLE_PROJECT_ID"
# Check the Google console for what cluster versions are available to use.
# This can found in "Master version" property when you start the creation of
# a cluster.
CLUSTER_VERSION="1.15.11-gke.3"
HELXPLATFORM_HOME=$HOME/src/helxplatform
CLUSTER_NAME="$USER-cluster"
# Copy "hydroshare-secret.yaml" to
#   "$HELXPLATFORM_HOME/hydroshare-secret.yaml" or set
#   HYDROSHARE_SECRET_SRC_FILE to point to it's location below, which is
#   currently the default value.
HYDROSHARE_SECRET_SRC_FILE="$HELXPLATFORM_HOME/hydroshare-secret.yaml"
```

(continues on next page)

(continued from previous page)

```
# The previous variables can also be exported instead of using a configuration
# file with GKE_CLUSTER_CONFIG exported below.
export GKE_CLUSTER_CONFIG=$HELXPLATFORM_HOME/env-vars-$USER-test-dev.sh

# Create directory to hold the source repositories.
mkdir -p $HELXPLATFORM_HOME
echo "export CLUSTER_NAME=$CLUSTER_NAME" > $GKE_CLUSTER_CONFIG
echo "export CLUSTER_VERSION=$CLUSTER_VERSION" >> $GKE_CLUSTER_CONFIG
echo "export PROJECT=$PROJECT" >> $GKE_CLUSTER_CONFIG
echo "export HYDROSHARE_SECRET_SRC_FILE=$HYDROSHARE_SECRET_SRC_FILE" >> $GKE_CLUSTER_
↪CONFIG

cd $HELXPLATFORM_HOME
git clone https://github.com/helxplatform/CAT_helm.git
git clone https://github.com/helxplatform/devops.git

cd $HELXPLATFORM_HOME/devops/bin
# To create the cluster using the config file specified as a command line
# argument run this.
./gke-cluster.sh -c $GKE_CLUSTER_CONFIG deploy all

# ...or with the GKE_CLUSTER_CONFIG variable exported you can just run this.
# ./gke-cluster.sh deploy all

# Work with cluster and then terminate it.
echo "###"
echo "When you are done with the cluster you can terminate it with"
echo "these commands."
echo "export GKE_CLUSTER_CONFIG=$HELXPLATFORM_HOME/env-vars-$USER-test-dev.sh"
echo "cd $HELXPLATFORM_HOME/devops/bin"
echo "./gke-cluster.sh delete all"
```

More Specific Installs

Setup a Kubernetes Cluster

Step-1: Before setting up a Kubernetes cluster we need to enable the Kubernetes Engine API.

Step-2: You can either using web based terminal provided by google(Google Cloud shell) or run the required command line interfaces on your own computers terminal.

Step-3: Ask Google cloud to create a "Kubernetes cluster" and a "default" node pool to get nodes from. Nodes represent the hardware and node pools will keep track how much of a certain type of hardware is required.

```
gcloud container clusters create \
  --machine-type n1-standard-2 \
  --image-type ubuntu \
  --num-nodes 2 \
  --zone us-east4-b \
  --cluster-version latest \
```

To check if the cluster is initialized,

```
kubectl get node -o wide
```

Step-4: Give your account permissions to grant access to all the cluster-scoped resources(like nodes) and namespaced resources (like pods). RBAC (Role-based access control) should be used to regulate access to resources to a specific user based on the role.

```
kubectl create clusterrolebinding cluster-admin-binding \
  --clusterrole=cluster-admin \
  --user=
```

Create a "cluster-admin" role and a cluster role biding to bind the role to the user.

Setup NFS server

Installing NFS server

Step-1: The NFS server is backed by a google persistent disk. Make sure it exists prior to the deployment. If the persistent dosk does not exist, use the following command to create a disk on a google kubernetes cluster,

```
gcloud compute disks create gce-nfs-disk --size 10GB --zone us-central1-a
```

Step-2: Run the following command to deploy NFS server.

```
kubectl apply -R -f nfs-server
```

6.3.1.4 Indices and tables

- genindex
- modindex
- search

6.4 Current & Upcoming Releases

HeLx is alpha. This section outlines a few areas of anticipated focus for upcoming improvements.

6.4.1 Architecture

Semantic Search

Our team has a semantic search engine for mapping to research data sets based on full text search. We anticipate extending the Tycho metadata model to include semantic links to ontologies, incorporating analytical tools into the semantic search capability. See EDAM & Biolink for more information.

Utilization Metrics

Basic per application resource utilization information is already provided. But we anticipate creating scalable policy based resource management able to cope with the range of implications of the analytic workspaces we provide, ranging from single user notebooks, to GPU accelerated workflows, to Spark clusters.

Proxy Management

Coming soon.

Helm 3 Deployment

Coming soon.

6.4.2 Current Deployments

Application	Version	Cluster	Namespace
ambassador	quay.io/datawire/ambassador:1.1.1	helx-prod	helx
cs-appstore	heliumdatastage/appstore:masterccav0.0.16	helx-prod	helx
nfs-server	gcr.io/google_containers/volume-nfs:0.8	helx-prod	helx
nginx-revproxy	heliumdatastage/nginx:cca-v0.0.5	helx-prod	helx
tycho-api	heliumdatastage/tycho-api:masterccav0.0.10	helx-prod	helx
ambassador	quay.io/datawire/ambassador:1.1.1	helx-val	default
nginxrevproxy	heliumdatastage/nginx:cca-v0.0.5	helx-val	default
ambassador	quay.io/datawire/ambassador:1.1.1	helx-dev	helx-dev
cs-appstore	heliumdatastage/appstore:masterccav0.0.18helx-dev	helx-dev	helx-dev
nfs-server	gcr.io/google_containers/volume-nfs:0.8	helx-dev	helx-dev
nginx-revproxy	heliumdatastage/nginx:cca-v0.0.5	helx-dev	helx-dev
tycho-api	heliumdatastage/tycho-api:masterccav0.0.12	helx-dev	helx-dev
ambassador	quay.io/datawire/ambassador:1.1.1	scidas-dev	default
cs-appstore	heliumdatastage/appstore:mastercca-v0.0.5	scidas-dev	default
nfs-server	gcr.io/google_containers/volume-nfs:0.8	scidas-dev	default

6.5 Indices and tables

- genindex
- modindex
- search